Lecture 4:

Miscellaneous Features

Parameterised Data Types

- Fortran 77 had a problem with numeric portability, the precision (and exponent range) between processors could differ,

- Fortran 90 implements a portable precision selecting mechanism,

- intrinsic types can be parameterised by a kind value (an integer). For example,

```
INTEGER(KIND=1) :: ik1
REAL(4) :: rk4
```

- the kind parameters correspond to differing precisions supported by the compiler (details in the compiler manual).

- objects of different kinds can be mixed in arithmetic expressions but procedure arguments must match in type **and** kind.

Integer Data Type by Kind

- selecting kind, by an explicit integer is still **not** portable,

- must use the `SELECTED_INT_KIND` intrinsic function. For example, `SELECTED_INT_KIND(2)` returns a kind number capable of expressing numbers in the range, $(-10^2, 10^2)$.

- here the argument specifies the minimum decimal exponent range for the desired model. For example,

```
INTEGER :: short, medium, long, vlong
PARAMETER (short = SELECTED_INT_KIND(2), &
           medium= SELECTED_INT_KIND(4), &
           long  = SELECTED_INT_KIND(10),&
           vlong = SELECTED_INT_KIND(100))
INTEGER(short)    :: a,b,c
INTEGER(medium)   :: d,e,f
INTEGER(long)     :: g,h,i
```

Constants of Selected Integer Kind

- Constants of a selected kind are denoted by appending underscore followed by the kind number or an integer constant name (better):

```
100_2, 1238_4, 54321_long
```

- Be **very careful** not to type a minus sign '-' instead of an underscore '_'!

- There are other pitfalls too, the constant

```
1000_short
```

  may not be valid as `KIND = short` may not be able to represent numbers greater than 100. Be very careful.

Real KIND Selection

Similar principle to INTEGER:

- SELECTED_REAL_KIND(8,9) will support numbers with a precision of 8 digits and decimal exponent range from $(-9, 9)$. For example,

  ```
  INTEGER, PARAMETER  ::
       r1 = SELECTED_REAL_KIND(5,20),  &
       r2 = SELECTED_REAL_KIND(10,40)
  REAL(KIND=r1)        :: x, y, z
  REAL(r2), PARAMETER :: diff = 100.0_r2
  ```

- COMPLEX variables are specified in the same way,

  ```
  COMPLEX(KIND=r1) :: cinema
  COMPLEX(r2) :: inferiority = &
             (100.0_r2,99.0_r2)
  ```

  Both parts of the complex number have the same numeric range.

Kind Functions

- it is often useful to be able to interrogate an object to see what kind parameter it has.
- `KIND` returns the integer which corresponds to the kind of the argument.
- for example, `KIND(a)` will return the integer parameter which corresponds to the kind of `a`. `KIND(20)` returns the kind value of the default integer type.
- the intrinsic type conversion functions have an optional argument to specify the kind of the result, for example,

```
print*, INT(1.0,KIND=3), NINT(1.0,KIND=3)
x = x + REAL(j,KIND(x))
```

Mixed Kind Expression Evaluation

Mixed kind expressions:

- If all operands of an expression have the same type and kind, then the result also has this type and kind.
- If the kinds are different, then operands with lower range are promoted before operations are performed. For example, if

  ```
  INTEGER(short) :: members, attendees
  INTEGER(long)  :: salaries, costs
  ```

  the expression:

  - `members + attendees` is of kind `short`,
  - `salaries - costs` is of kind `long`,
  - `members * costs` is also of kind `long`.

- Care must be taken to ensure the LHS is able to hold numbers returned by the RHS.

Kinds and Procedure Arguments

Dummy and actual arguments must match exactly in kind, type and rank, consider,

```
SUBROUTINE subbie(a,b,c)
 USE kind_defs
 REAL(r2), INTENT(IN)  :: a, c
 REAL(r1), INTENT(OUT) :: b
  ...
```

an invocation of `subbie` must have matching arguments, for example,

```
 USE kind_defs
 REAL(r1) :: arg2
 REAL(r2) :: arg3
  ...
 CALL subbie(1.0_r2, arg2, arg3)
```

Using `1.0` instead of `1.0_r2` will not be correct on every compiler.

This is very important with generics.

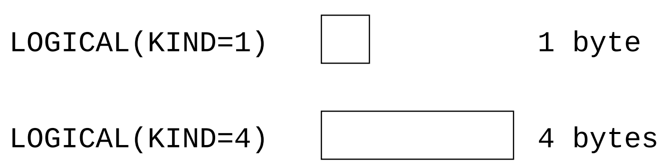
Logical KIND Selection

- There is no `SELECTED_LOGICAL_KIND` intrinsic, however, the `KIND` intrinsic can be used as normal.

  For example,

```
LOGICAL(KIND=4) :: yorn = .TRUE._4
LOGICAL(KIND=1), DIMENSION(10) :: mask
IF (yorn .EQ. LOGICAL(mask(1),KIND(yorn)))...
```

- `KIND=1` may only use one byte of store per variable,



- Must refer to the compiler manual.

Character KIND Selection

- Every compiler must support at least one character set which must include all the Fortran characters. A compiler may also support other character sets:

```
INTEGER, PARAMETER :: greek = 1
CHARACTER(KIND=greek) :: zeus, athena
CHARACTER(KIND=2,LEN=25) :: mohammed
```

- Normal operations apply individually but characters of different kinds cannot be mixed. For example,

```
print*, zeus//athena   ! OK
print*, mohammed//athena ! illegal
print*, CHAR(ICHAR(zeus),greek)
```

  Note CHAR gives the character in the given position in the collating sequence.

- Literals can also be specified:

  greek_"$\alpha\delta\alpha\mu$"

  Notice how the kind is specified first.

Mathematical Intrinsic Functions

Summary,

`ACOS(x)`	arccosine
`ASIN(x)`	arcsine
`ATAN(x)`	arctangent
`ATAN2(y,x)`	arctangent of complex number (x, y)
`COS(x)`	cosine where x is in radians
`COSH(x)`	hyperbolic cosine where x is in radians
`EXP(x)`	e raised to the power x
`LOG(x)`	natural logarithm of x
`LOG10(x)`	logarithm base 10 of x
`SIN(x)`	sine where x is in radians
`SINH(x)`	hyperbolic sine where x is in radians
`SQRT(x)`	the square root of x
`TAN(x)`	tangent where x is in radians
`TANH(x)`	tangent where x is in radians

Numeric Intrinsic Functions

Summary,

`ABS(a)`	absolute value
`AINT(a)`	truncates `a` to whole `REAL` number
`ANINT(a)`	nearest whole `REAL` number
`CEILING(a)`	smallest `INTEGER` greater than or equal to `REAL` number
`CMPLX(x,y)`	convert to `COMPLEX`
`DBLE(x)`	convert to `DOUBLE PRECISION`
`DIM(x,y)`	positive difference
`FLOOR(a)`	biggest `INTEGER` less than or equal to real number
`INT(a)`	truncates a into an `INTEGER`
`MAX(a1,a2,a3,...)`	the maximum value of the arguments
`MIN(a1,a2,a3,...)`	the minimum value of the arguments
`MOD(a,p)`	remainder function
`MODULO(a,p)`	modulo function
`NINT(x)`	nearest `INTEGER` to a `REAL` number
`REAL(a)`	converts to the equivalent `REAL` value
`SIGN(a,b)`	transfer of sign — `ABS(a)*(b/ABS(b))`

Character Intrinsic Functions

Summary,

`ACHAR(i)`	i^{th} character in ASCII collating sequence
`ADJUSTL(str)`	adjust left
`ADJUSTR(str)`	adjust right
`CHAR(i)`	i^{th} character in processor collating sequence
`IACHAR(ch)`	position of character in ASCII collating sequence
`ICHAR(ch)`	position of character in processor collating sequence
`INDEX(str,substr)`	starting position of substring
`LEN(str)`	Length of string
`LEN_TRIM(str)`	Length of string without trailing blanks
`LGE(str1,str2)`	lexically `.GE.`
`LGT(str1,str2)`	lexically `.GT.`
`LLE(str1,str2)`	lexically `.LE.`
`LLT(str1,str2)`	lexically `.LT.`
`REPEAT(str,i)`	repeat i times
`SCAN(str,set)`	scan a string for characters in a set
`TRIM(str)`	remove trailing blanks
`VERIFY(str,set)`	verify the set of characters in a string

Bit Manipulation Intrinsic Functions

Summary,

`BTEST(i,pos)`	bit testing
`IAND(i,j)`	AND
`IBCLR(i,pos)`	clear bit
`IBITS(i,pos,len)`	bit extraction
`IBSET(i,pos)`	set bit
`IEOR(i,j)`	exclusive OR
`IOR(i,j)`	inclusive OR
`ISHFT(i,shft)`	logical shift
`ISHFTC(i,shft)`	circular shift
`NOT(i)`	complement
`MVBITS(ifr,ifrpos, len,ito,itopos)`	move bits (SUBROUTINE)

Variables used as bit arguments must be `INTEGER` valued. The model for bit representation is that of an unsigned integer, for example,

```
s-1        3  2  1  0
 0  ..  |  0  0  0  0    value = 0

s-1        3  2  1  0
 0  ..  |  0  1  0  1    value = 5

s-1        3  2  1  0
 0      |  0  0  1  1    value = 3
```

The number of bits in a single variable depends on the compiler

Array Construction Intrinsics

There are four intrinsics in this class:

- ☐ `MERGE(TSOURCE,FSOURCE,MASK)`— merge two arrays under a mask,
- ☐ `SPREAD(SOURCE,DIM,NCOPIES)`— replicates an array by adding `NCOPIES` of a dimension,
- ☐ `PACK(SOURCE,MASK[,VECTOR])`— pack array into a one-dimensional array under a mask.
- ☐ `UNPACK(VECTOR,MASK,FIELD)`— unpack a vector into an array under a mask.

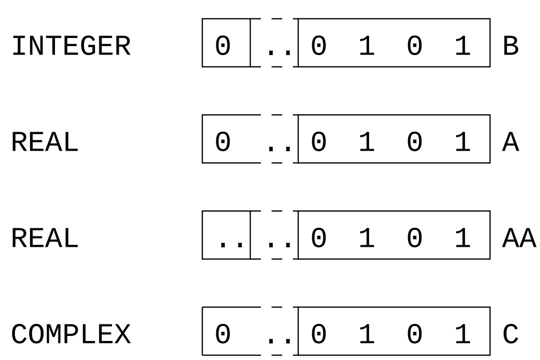
TRANSFER Intrinsic

TRANSFER converts (not coerces) physical representation between data types; it is a retyping facility. Syntax:

TRANSFER(SOURCE,MOLD)

- SOURCE is the object to be retyped,
- MOLD is an object of the target type.

```
REAL, DIMENSION(10)    :: A, AA
INTEGER, DIMENSION(20) :: B
COMPLEX, DIMENSION(5)  :: C
 ...
A  = TRANSFER(B, (/ 0.0 /))
AA = TRANSFER(B, 0.0)
C  = TRANSFER(B, (/ (0.0,0.0) /))
 ...
```



Fortran 95

Fortran 95 will be the new Fortran Standard.

- `FORALL` statement and construct

```
FORALL(i=1:n:2,j=1:m:2)
 A(i,j) = i*j
END FORALL
```

- nested `WHERE` constructs,
- `ELEMENTAL` and `PURE` procedures,
- user-defined functions in initialisation expressions,
- automatic deallocation of arrays,
- improved object initialisation,
- remove conflicts with IEC 559 (IEEE 754/854) (floating point arithmetic),
- deleted features, for example, `PAUSE`, assigned `GOTO`, `cH` edit descriptor,
- more obsolescent features, for example, fixed source form, assumed sized arrays, `CHARACTER*`< *len* > declarations, statement functions,
- language tidy-ups and ambiguities (mistakes),

High Performance Fortran

High Performance Fortran (or HPF) is an ad-hoc standard based on Fortran 90. It contains

- Fortran 90,
- syntax extensions, `FORALL`, new intrinsics, `PURE` and `ELEMENTAL` procedures,
- discussion regarding storage and sequence association,
- compiler directives:

```
!HPF$ PROCESSORS P(5,7)
!HPF$ TEMPLATE T(20,20)
      INTEGER, DIMENSION(6,10) :: A
!HPF$ ALIGN A(J,K) WITH T(J*3,K*2)
!HPF$ DISTRIBUTE T(CYCLIC(2),BLOCK(3)) ONTO P
```

Data Alignment